

Logistics Simulator Report

- 该项目是基于Python语言进行的物流公司运输模拟，在其中用到的数据结构均为Python自带或调用Python相关开源包。在运行该项目代码前，请阅读**使用指引**。
- 此外，在本项目中，我们固定station和center的数量为25和5。并且在需要将这两者存在同一个矩阵或链表中时，前25项为station，后5项为center。
- 共有五个文件夹，四个为模型，一个为模型+代价评估，其中：Basic文件夹下是Time-Driven模型，Event-Driven文件夹下是Event-Driven的基础模型，Route_optimize文件夹下是加入Route_optimized的Event-Driven模型，Queue文件夹下是加入Queue_optimized的Event-Driven模型，Cost文件夹下是代价评估以及对应模拟。

基于Time-Driven的模型

- 对于每个站点的throughput，在本模型中，我们均认为是，在time_tick=1这样一个时刻中，站点能够处理的包裹数量，比如，如果s1的throughput=8，那么它在t=0~1, 1~2, 2~3...中最多可以处理8个包裹。

1. 数据初始化

首先根据题目，我们创建了三个类：Location, Route 和 Package

```
1 # Location
2 class Location:
3     def __init__(self, ID, pos, throughput, delay, cost):
4         self.ID = ID
5         self.pos = pos
6         self.throughput = throughput
7         self.delay = delay
8         self.cost = cost
9         self.buffer = {'arrived': [], 'processing': [], 'ready_to_send': []}
10        self.pack_passby = PriorityQueue()
11        self.capa = self.through
```

- 在Location类中，存储的是station和center。其中ID, pos, throughput, delay, cost 均为默认的data_gen随机生成。此外又对每个location维护了一个buffer池，该池为一个字典，分别为"arrived"：存放到达该站但是还没有进行process的包裹；"processing"：存放到达该站已经完成排队并开始processing的包裹；"ready_to_send"：存放已经完成processing，等待被发出的包裹。pass_by是一个优先队列，即最小堆，用来按到达时间（key）存放到达该站的包裹的ID（value），并且方便按时间顺序弹出。capa是用来记录在某一time_tick下，该站点还能处理多少包裹。

```

1 # Route
2 class Route:
3     def __init__(self, src, dst, time, cost):
4         self.src = src
5         self.dst = dst
6         self.time = time
7         self.cost = cost
8 # Package
9 class Package:
10     def __init__(self, ID, time_created, src, dst, category):
11         self.ID = ID
12         self.time_created = time_created
13         self.src = src
14         self.dst = dst
15         self.category = category

```

- 在Route类中，存储的是道路信息，其中所有信息均为data_gen随机生成。
- 在Package类中，存储的是包裹的信息，其中所有信息均为data_gen随机生成。

接下来，我们创建了一个新的类：simulator，用于进行模拟，存储模拟以及需要用到的函数。

```

1 class simulator:
2     def __init__(self, data):
3         self.locations = self.setlocation(data)
4         self.routes = self.setroutes(data)
5         self.packages = self.setpackages(data)
6         self.G = self.route_graph(data) # 把route转换成graph, 用去年图论的nx包
7         self.tracking_info = {}
8         self.on_route = [] # 在路上的包裹
9         self.route_time_matrix = self.matrix("time") # 路线的时间矩阵
10        self.route_cost_matrix = self.matrix("cost") # 路线的成本矩阵
11        self.route_min_time = self.chemin_time_matrix() # 最短时间路线矩阵
12        self.route_min_cost = self.chemin_cost_matrix() # 最小成本路线矩阵

```

```
13         self.packageID = [str(x.ID) for x in self.packages]
14         self.fig, self.ax = self.bg(data)
```

- 在初始化中，`locations` 是将data信息转换成Location类并存储在一个链表中，`routes` 是将data信息中的路径转换成Route类并存储在一个链表中，`packages` 是将data信息中的包裹转换成Package类并存储在一个链表中。`G` 则是把所有路径看做edges从而将整个地图信息变成一个加权的graph，方便后续进行可视化等。`route_time_matrix` 是把路径信息转换为一个邻接矩阵，其中存储站点到站点之间的路径时间；`route_cost_matrix` 是把路径信息转换为一个邻接矩阵，其中存储站点到站点之间的成本时间。`self.fig, self.ax` 则是继承了data_gen中的map背景，方便后续画图。`packageID` 是存储所有包裹的ID的链表。`tracking_info` 为一个字典，键为包裹的UUID，值为包裹的一系列运输信息。`on_route` 则是所有在路径上的包裹的信息，即被站点发出但是还未到下一个目的地的包裹信息。有关以上这些的初始化函数不再赘述。
- 接下来是对 `route_min_time` 和 `route_min_cost` 的解释：

在基础模型里，我们计算包裹的最优路径（成本最低或时间最短）时，只考虑一开始生成的路径信息，不考虑当前时刻的排队等情况，包裹的最优路径被认为是固定的（即在任何时刻，从某站点到另一站点的最优路径都一样），并且在之后的运输中不再改变。

根据这个，我们希望创建一个矩阵，在其中存储各个站点之间的最优路径，这样在以后要求最优路径时，只需要调用该矩阵即可。以express包裹的最短路径矩阵 `route_min_time` 为例，我们遍历每个站点，利用dijkstra算法求得该站点到其他所有站点的最短路径，并把这些路径信息存在一个 `route_min_time` 矩阵中，也就是说，在这个矩阵中，它的第2行第8列就代表s2到s8的最短路径。`route_min_cost` 同理。

所以，根据上述过程，我们创建了两个函数直接用来提取最优路径（返回最优路径，类型为列表）：

```
1  # 这个函数是准备用来处理express包裹的
2  def pack_exp(self, src, dst):
3      #chemin_min = self.chemin_time_matrix()
4      src = int(src[1:])
5      dst = int(dst[1:])
6      route_min = self.route_min_time[src][dst]
7      route_min_new = []
8      for i in range(len(route_min)):
9          route_min_new.append(self.int_ID(route_min[i]))
10     return route_min_new
11
12  # 处理standard包裹
13  def pack_normal(self, src, dst):
14      #chemin_min = self.chemin_cost_matrix()
```

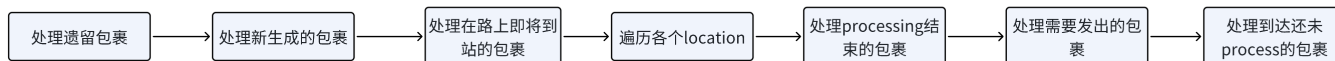
```

15     src = int(src[1:])
16     dst = int(dst[1:])
17     route_min = self.route_min_cost[src][dst]
18     route_min_new = []
19     for i in range(len(route_min)):
20         route_min_new.append(self.int_ID(route_min[i]))
21     return route_min_new

```

2. 模拟过程

- 该实现方式是以离散的时间进行循环和遍历来做的，即从t=1开始，先处理t=0~1期间发生的所有事件，然后进行t=2，处理t=1~2之间发生的所有事件.....直到所有包裹都到达目的地。
- 在开始循环之前，我们先创建了一个新的变量：`pack_Delivered = parameters["packet_num"]` 用来存储目前没有到站的包裹数量，当这个数量为0时，结束循环。
- 接下来是在每个时间节点的处理过程：



2.1 处理遗留包裹

先遍历所有站点，看站点的 `buffer['arrived']` 中是否还有遗留包裹未处理，即排队的包裹。`buffer['arrived']` 中的元素为 `[Package(class), arrive_time, left_route]` (Package类，到站时间，剩余路径)。首先更新该站点的capa值，将其恢复到throughput。然后对 `buffer['arrived']` 按时间顺序进行排序，如果capa不等于0且 `buffer['arrived']` 非空，则进行以下循环：

- 取 `buffer['arrived']` 第一个元素，计算它的等待时间，并将 `[Package(class), t, left_route]` (Package类，目前时间（即该包裹开始处理的时间），剩余路径) 添加至 `buffer["processing"]` 中。
- 将该站点的capa减去1，并将该包裹信息从 `buffer["arrive"]` 删除。

🍒 比如在s2的 `buffer['arrived']` 中，依次有`[[pack1, 13.45, [c1,s5]], [pack2, 14.53,[c2, s4]]`，这两个包裹均在排队，但是因为pack1到达时间比pack2早，所以先处理pack1，再处理pack2。

2.2 处理新生成的包裹

如果packages列表中还有元素，即仍有未生成的包裹，则提取出packages中的第一个包裹：

- 如果该包裹的 `create_time` 小于当前时间 `t`：
 - 首先根据它的类型为其分配路径，记在 `pack_chemin` 中，并把这个包裹的信息作为 `[Package(class), time_created, pack_chemin]` 列表添加到对应的Location类下的 `buffer["arrived"]` 中；
 - 把该包裹的创建时间和ID作为键值对插入到 `pass_by` 中；
 - 在 `tracking_info` 中创建新的键值对，键为包裹的UUID，值为包裹的运输信息：`[src, dst, time_created, np.inf, [time_created, src, "Arrived"]]`，依次为始发地，目的地，创建时间，到达时间，Log信息（Time, Location, Event）；
 - 最后删除该元素。
- 如果该包裹的 `create_time` 大于当前时间 `t`，则停止遍历。（因为在创建packages的时候是按创建时间放入包裹的）

🍷 比如当前时刻为`t=5`，packages中第一个包裹pack的UUID为`a123`，创建时间为`4.123`，始发地为`s2`，终点站为`s8`，类型为`1`，我们先调用`s2`到`s8`的时间最短路径，为`[s2, s5, c2, s8]`，然后把`[pack, 4.123, [s5, c2, s8]]`添加到`s5`站点的`buffer["arrived"]`中；然后在`tracking_info`中新建键为`a123`，值为`[s2, s8, 4.123, np.inf, [4.123, s2, "Arrived"]]`的键值对；最后从`packages`中删除该包裹。

2.3 处理在路上即将到站的包裹

`on_route` 列表中存放的是正在路径上的包裹，该列表中的元素为：`[Package(class), next_station, arrived_time, left_route]`，分别为Package类，即将到达的站点，到达时间，剩余路径。

遍历 `on_route` 这一列表，每次提取出一个包裹的信息：

- 如果该包裹的到达时间 `arrived_time` 小于当前时间 `t`：
 - 首先把这个包裹的信息作为 `[Package(class), arrived_time, left_chemin]`（Package类，到达时间，剩余路径）添加到对应的Location类下的 `buffer["arrived"]` 中；
 - 再把该包裹的到达时间和ID作为键值对插入到 `pass_by` 中；

- 然后在 `tracking_info` 中找到UUID键对应的值，并新增 `[arrived_time,next_station,"Arrived"]`（到达时间，到达站点，状态）；
 - 最后删除该元素。
- 如果该包裹的到达时间 `arrived_time` 大于当前时间 `t`，则跳过该包裹。

🍒 比如当前时刻为`t=19`，`on_route`中的一个包裹信息为 `[Package(class), s9, 18.354, [s9,s10]]`，那么就把 `[Package(class), 18.354, [s10]]` 添加到`s9`站点的 `buffer["arrived"]`；然后在 `tracking_info` 中找到键为该包裹ID的值，并在该列表后添加 `[18.354, s9, "Arrived"]`；最后从 `on_route` 中删除该包裹。

2.4 处理站内的buffer池

我们遍历 `locations` 中的所有Location类，我们来看该站点维护的三个不同buffer池

`["arrived"]` `["processing"]` `["ready_to_send"]`：

- 首先遍历 `buffer["processing"]`，其中的元素为 `[Package(class),start_time,left_route]`（Package类，开始process的时间，剩余路径）。
 - 如果该包裹开始process的时间加上该站点的delay时间，小于当前时间`t`，就把 `[Package(class),send_time,left_chemin]`（Package类，被发出的时间，剩余路径）添加到该站点的 `buffer["processing"]` 中；最后删除该元素。
 - 如果没有处理完毕，则跳过该包裹。

🍒 比如当前时刻为`t=19`，站点`s8`的 `buffer["processing"]` 中一个包裹信息为 `[Package(class), 15.354, [c1,s10]]`，`s8`的delay为3，那么该包裹已经完成process，将 `[Package(class), 18.354, [c1,s10]]` 添加到 `buffer["ready_to_send"]` 中；最后从 `buffer["processing"]` 删除该包裹。

- 接下来遍历 `buffer["ready_to_send"]`，其中元素为 `[Package(class),send_time,left_route]`（Package类，被发出的时间，剩余路径），先将 `[send_time, loc_id, "Sent"]`（发出时间，站点，状态）添加到 `tracking_info` 中该包裹对应的值中。
 - 如果该包裹的剩余路径为空集（即已经到达终点站），那么就把 `tracking_info` 中该包裹的到达时间从 `np.inf` 修改为 `send_time`，并让 `pack_Delivered-1`；

- 如果剩余路径不为空，那么该包裹需要继续走，然后将 `[Package(class), next_loc, time_arrive, left_route]` (Package类，下一站，到达时间，剩余路径) 添加到 `on_route` 中。最后删除该元素。

🍒 比如当前时刻为`t=19`，站点`s8`的 `buffer["ready_to_send"]` 中一个包裹信息为 `[Package(class), 18.354, [c1,s10]]`，根据剩余路径可知，下一站为`c1`，然后根据之前创建的矩阵可知，`s8`到`c1`的时间为6，先在 `tracking_info` 中找到键为该包裹ID的值，并在该列表后添加 `[18.354, s8, "Sent"]`，再将 `[Package(class), c1, 24.354, [c1,s10]]` 添加到 `on_route` 中，最后从 `buffer["ready_to_send"]` 删除该包裹。

如果该包裹信息为 `[Package(class), 18.354, []]`，则该包裹已经到达终点站，那么先在 `tracking_info` 中找到键为该包裹ID的值，并在该列表后添加 `[18.354, s8, "Sent"]`，并把到达时间改为 `18.354`，然后将 `pack_Delivered-1`，最后从 `buffer["ready_to_send"]` 删除该包裹。

- 最后看 `buffer["arrived"]`，其中元素为 `[Package(class), arrive_time, left_route]` (Package类，到达时间，剩余路径)。如果`capa`不等于0且 `buffer['arrived']` 非空（说明没有任何在排队的包裹，到站的包裹可以立刻开始处理），则进行以下循环：
 - 取 `buffer['arrived']` 第一个元素，计算它的等待时间，并将 `[Package(class), arrive_time, left_route]` (Package类，到站时间（即包裹开始处理的时间），剩余路径) 添加至 `buffer["processing"]` 中。
 - 将该站点的`capa`减去1，并将该包裹信息从 `buffer["arrive"]` 删除。

2.5 循环结束

在完成以上三个过程后，将`t+1`，进行判断：`pack_Delivered = 0?`，根据结果判断循环是否继续，如需继续，则跳回步骤1；如果不需要进行循环，则整个模拟进程结束。

基于Event-Driven的基础模型（1）

- 在这个模型里，基本思想和Time-Driven一致，即包裹路径不做改变，但是遍历方式从离散的时间驱动，改为了由物流系统中的事件驱动。

- 对于每个站点的throughput，在本模型中，我们均认为是，在time_tick=1这样一个时刻中，站点能够处理的包裹数量，比如，如果s1的throughput=8，那么它在t=0~1, 1~2, 2~3...中最多可以处理8个包裹。

1. 数据初始化

对 Location, Route 和 Package 类的构建基本与Time-Driven一样，在这里讲一下有更改的地方。

```
1 # Location
2 class Location:
3     def __init__(self, ID, pos, throughput, delay, cost):
4         self.ID = ID
5         self.pos = pos
6         self.throughput = throughput
7         self.delay = delay
8         self.cost = cost
9         self.pack_passby = PriorityQueue()
10        self.capa = [throughput, 0]
11
12        # 更新timetick和处理量
13        def update(self, t):
14            if self.capa[1] < int(t):
15                self.capa[1] = int(t)
16                self.capa[0] = self.throughput
```

- Location类里更改了 `capa` 变量，它是一个由两个值组成的列表，它的第二项是当前事件的时间，被初始化为0；第一项是在这个时间节点中还能处理多少包裹，被初始化为该站点的throughput；并新增了一个 `update` 函数用来更新capa的值。下面是对 `capa` 的解释：如果 `capa=[5, 32.34]`，则说明该站点在t=32~33时刻里，最多可以再将5个包裹开始process。关于 `update`：需要处理包裹前调用，如果当前时间与上次处理时间不在同一个time_tick，则恢复capa为throughput。

```
1 # Package
2 class Package:
3     def __init__(self, ID, time_created, src, dst, category):
4         self.ID = ID
5         self.time_created = time_created
6         self.src = src
7         self.dst = dst
8         self.category = category
9         self.tracking_info = []
```



```

10         self.arrived_time = 0
11
12     def __lt__(self, other):
13         return self.arrived_time < other.arrived_time

```

- Package类中更新了 `arrived_time` 变量，用于记录该包裹到达下一目的地的时间。`__lt__`函数则是方便后续对Package类进行比大小，在事件发生的时间相同时，会优先取出`arrived_time`即更早到达站点的包裹。

在simulator类的初始化中，也做了一些改动，同样也只在这里讲一下有更改的地方。

```

1 class simulator:
2     def __init__(self, data):
3         # 导入模拟的初始数据 (input)
4         self.locations = self.setlocation(data)
5         self.routes = self.setroutes(data)
6         self.packages = self.setpackages(data)
7         self.G = self.route_graph(data) # 把route转换成graph, 用去年图论的nx包
8         self.tracking_info = {}
9         self.route_time_matrix = self.matrix("time") # 路线的时间矩阵
10        self.route_cost_matrix = self.matrix("cost") # 路线的成本矩阵
11        self.route_min_cost = self.chemin_cost_matrix() # 最小成本路线矩阵
12        self.route_min_time = self.chemin_time_matrix()
13        self.events = self.init_events()
14        self.packageID = [str(x.ID) for x in self.packages]
15        self.fig, self.ax = self.bg(data)

```

- 删除了 `on_route`。
- 新增了 `events` 池，这是一个最小堆，键为事件的时间，值为包裹的时间信息和状态，用于记录整个模拟流程中的事件。
 - 在该模型中，共有四种不同的事件：Created（包裹被创建），Arrived（包裹到达站点），Waiting（包裹等待被处理），toSend（包裹等待被发出）。
 - 该池在初始化时，将所有包裹均标记为Created，并按创建时间将 `[created_time, Package(class), "Created"]` 放入事件池中，最后将所有包裹的 `arrived_time` 更改为创建时间。

2. 模拟过程

- 因为events池是一个最小堆，所以每次取最上面的节点，都是整个流程中下一步最先发生的事件。event-driven就是基于这个来实现的，即每次都弹出events的最上面的事件，判断这个包裹处于什么状态后，针对这个状态进行处理，直到该事件池为空（所有包裹都已经到达终点站）。
- 下面是对于每个包裹状态的处理过程：

2.1 状态为"Created"

如果包裹的状态为"Created"，即该包裹刚被创建，可以通过Package类找到该包裹的起始点和终点，并通过函数直接调出它的最优路径。

- 在 `tracking_info` 中创建新的键值对，键为包裹的UUID，值为包裹的运输信息：`[src, dst, time_created, np.inf, [time_created, src, "Arrived"]]`，依次为始发地，目的地，创建时间，到达时间，Log信息（Time, Location, Event）；
- 把该包裹的创建时间和ID作为键值对插入到 `pass_by` 中；
- 接下来我们需要检查这个包裹是否能够在这个站点立刻开始process流程，在这里需要先调用Location类中的 `update` 函数，更新目前时刻该站点的 `capa`，即该站点还目前能够处理多少包裹：
 - 如果 `capa[0]` 不为0，即包裹无需等待可立刻开始处理：
 - 在 `tracking_info` 中找到UUID键对应的值，并新增 `[time_created, loc_id, "Processing"]`（开始process的时间，站点，状态）；
 - 在events里插入 `[time_created+delay, loc_id, pack_route, "toSend"]`（处理完毕可以发出的时间，站点，包裹路径，状态）；
 - 将 `capa[0]` 减去1，表示该time_tick下可处理包裹减少1。
 - 如果 `capa[0]` 为0，即包裹需要排队：
 - 在events里插入 `[int(time_created)+1, loc_id, pack_route, "Waiting"]`（下次轮到时间，站点id，包裹路径，状态），`int(time_created)+1` 是因为在下一个time tick开始时，站点的处理量会恢复，需要先处理正在排队的包裹。



假如目前的事件为 `[13.44, pack, s4, "Created"]`，那么我们现在在 `tracking_info` 中创建一个键，键的值为它的ID，并填上其他信息；随后检查目前s4站点还有多少capa比如s4的throughput为10，delay=1：


- 如果此时的 `capa=[0, 12]`，就说明上次s4处理的包裹在t=12~13区间内，并且用完了所有吞吐量，则现在需要把capa更新为 `[10, 13]`，说明可以处理该包裹，需要在events里插入新事件：`[13.44+1, s4, route, "toSend"]`，并将capa的第一项减1，表示在这个t=13~14这个time tick中还能处理9个包裹。

- 如果此时的 `capa=[0, 13]`，就说明s4在这个time tick中已经不能再处理包裹，此时，这个包裹希望在s4恢复吞吐量时被有限处理，即t=14时开始处理（如果t=14需要处理的包裹不多），所以在events里插入新事件 `[14, s4, route, "Waiting"]`。

2.2 状态为"Waiting"

如果包裹的状态为"Waiting"，即该包裹正在排队。在这里需要先调用Location类中的 `update` 函数，更新目前时刻该站点的 `capa`，即该站点还目前能够处理多少包裹：

- 如果 `capa[0]` 不为0，即站点还可以继续处理包裹：
 - 在 `tracking_info` 中找到UUID键对应的值，并新增 `[time, loc_id, "Processing"]`（当前时间，站点，状态），在当前时间就可以开始处理；
 - 在events里插入 `[time+delay, loc_id, pack_route, "toSend"]`（处理完毕可以发出的时间，站点，包裹路径，状态）；
 - 将 `capa[0]` 减去1，表示该time_tick下可处理包裹减少1。
- 如果 `capa[0]` 为0，即包裹还需要排队：
 - 在events里插入 `[int(time_created)+1, loc_id, pack_route, "Waiting"]`（下次轮到时间，站点id，包裹路径，状态），`int(time_created)+1` 是因为在下一个time tick开始时，站点的处理量会恢复，需要先处理正在排队的包裹。

 在events进行弹出的时候，如果有两个事件的时间都为t，那么会继续比较这两个包裹的到站时间，到站更早的会被弹出。

2.3 状态为"toSend"

如果包裹的状态为"Waiting"，即该包裹已经处理完毕等待被发送。

- 首先，在 `tracking_info` 中找到UUID键对应的值，并新增 `[time, loc_id, "Sent"]`（当前时间，站点，状态）；
- 然后判断该包裹是否已经到达终点，即剩余路径是否为空：
 - 如果已经到达终点，那么就把 `tracking_info` 中该包裹的到达时间从 `np.inf` 修改为 `time`（当前时间），并让 `pack_Delivered-1`；
 - 如果还未到达终点，计算它到达下一站的时间，将 `[time_arrive, Package(class), next_loc, left_route, "Arrived"]`（到达时间，Package类，下一站，剩余路径，状态）插入到 `events` 中。

2.4 状态为"Arrived"

如果包裹的状态为"Arrived"，即该包裹到达了当前目的地。

- 在 `tracking_info` 中找到UUID键对应的值，并新增 `[time, loc_id, "Arrived"]`（当前时间，站点，状态）；
- 把该包裹的到达时间和ID作为键值对插入到 `pass_by` 中；
- 调用Location类中的 `update` 函数，更新目前时刻该站点的 `capa`，即该站点还目前能够处理多少包裹：
 - 如果 `capa[0]` 不为0，即包裹无需等待可立刻开始处理：
 - 在 `tracking_info` 中找到UUID键对应的值，并新增 `[time, loc_id, "Processing"]`（当前时间，站点，状态），在当前时间就可以开始处理；
 - 在events里插入 `[time+delay, loc_id, pack_route, "toSend"]`（处理完毕可以发出的时间，站点，包裹路径，状态）；
 - 将 `capa[0]` 减去1，表示该time_tick下可处理包裹减少1。
 - 如果 `capa[0]` 为0，即包裹需要排队：
 - 在events里插入 `[int(time_created)+1, loc_id, pack_route, "Waiting"]`（下次轮到时间，站点id，包裹路径，状态），`int(time_created)+1` 是因为在下一个time tick开始时，站点的处理量会恢复，需要先处理正在排队的包裹。

2.5 循环结束

每次处理完当前事件后，进行判断：`pack_Delivered = 0?`，根据结果判断循环是否继续，如需继续，则跳回步骤1；如果不需要进行循环，则整个模拟进程结束。

3. 与Time-Driven的对比

- Event-driven的实现更符合逻辑，并且相较之下代码实现更加简单和简洁，每次更新只需要局部优化，对特定站点进行处理；并且脱离了discrete-time，可以保持各个站点时间上的统一，更好地应对复杂情况，或进行道路的优化。
- 同时，对于需要排队的情况，time-driven有一定缺陷。比如，当s1和s2都有一个排队包裹（分别记作pack1和pack2），需要在t=25时被处理，假设pack1的到达时间为19.87，pack2的到达时间为17.88，在这种情况下，很明显对于整个场景，我们应该先处理pack2，但是由于遍历站点的时候先遍历到s1再遍历到s2，所以会导致在time-driven下，pack1会被先处理；如果想解决这一情况，需要不断遍历所有站点的buffer池子，找到其中最早到达的包裹再进行处理，这样会大大提高模型的时间成本
- 但是Event-driven的实现也有缺点，在相同包裹数量下，可以发现event-driven更加耗时，因为在维护事件池时使用的是最小堆，基于平衡二叉树，每次插入和删除的成本均为 $O(\log n)$ ，在包裹数量很大的时候插入和删除效率低下，所以时间更长。

--	--	--	--	--


```

26         break
27         num_left = max(0, num_left-t*self.throughput)
28         waiting_t = num_left // self.throughput
29         return waiting_t

```

- 在Location类中，更新了一个pack_arrive的链表，存放已经发出即将到达该站点的包裹；以及一个buffer_num，用来存储目前这个站点在排队的包裹数量。
- 此外，加入了一个新的函数 `waiting_At_t`，用于计算从当前时间经过t时刻后，该站点需要排队的时间。

2. 模拟过程

因为只有express包裹是需要找最短路径的，所以在路径优化时，只限于更改express包裹的路径，而不改变standard包裹的默认路径。

我们优化的主要思路为：

- express包裹的路径不再固定，而是每向下走一步就更新一次最短路径。
- 重新计算包裹的最短路径，即在每次找到状态为"toSend"的express包裹时：
 - 统计场上所有站点的情况：即将到站的包裹 `pack_arrive` 和正在该站点排队的包裹数量 `buffer_num`。
 - 根据上面的情况，局部更新邻接矩阵：如果站点i可以到达站点j，且时间为t，计算在t时刻时，该站点是否需要排队，即看 `waiting_At_t` 函数的输出是否大于0，如果需要排队并且排队时间为t1，那我们认为这条路径现在所需的时间为t1+t，所以在邻接矩阵中站点i到站点j的时间更改为t1+t。

```

1 def update_M(self, tt):
2     M_new = self.route_time_matrix.copy()
3     for i1 in range(len(self.route_time_matrix)):
4         x = self.route_time_matrix[i1]
5         for i2 in range(len(x)):
6             if x[i2] != np.inf:
7                 time_route = x[i2]
8                 time_waiting =
self.locations[i2].waiting_At_t(tt+time_route)
9                 if time_waiting > 0:
10                     M_new[i1][i2] = time_waiting+time_route
11     return M_new

```

- 通过上述方式更新后的邻接矩阵，再利用dijkstra算法，计算该包裹的最短路径，让该包裹按新路径继续向下走。

- 重新计算包裹的最短路径，即在每次找到状态为"toSend"的express包裹时：

3. 与基础模型的对比

优点：

- 整个系统的优化效果比较明显，以5000个包裹为例，对比基础模型，有2000多个包裹运送的更快。
- 运送变快的包裹有express的包裹也有standard的包裹，在规避了express包裹的排队后，standard包裹的排队时间也减少了，整个流程节省了2000~8000个time tick。

下面是一个output：

```
1 The total number of slower packets is 1522.
2 The total number of faster packets is 2827.
3 Saved time: 4485.245923689581
4 The average saved time of faster packet is 30.11703739705968.
```

```
1 The total number of slower packets is 2135.
2 The total number of faster packets is 2946.
3 Saved time: 5571.1589756261678
4 The average saved time of faster packet is 37.89462251427518.
```

不足：

- 因为计算路径时只能对于当前的第一步预测比较有效，整个系统还是在不断变化的，当这个包裹走完一步后，下一步的情况很难预测，有可能在当前情况下，第二步是最优的，但是在后续有别的包裹陆续发出后，该路径会变得更慢。
- 这个算法的时间复杂度太高，在每次发出包裹时都需要重新更新邻接矩阵并使用dijkstra算法，所以dijkstra会比基础模型多调用几万次，所以导致整体运行时间特别长。

基于Event-Driven的基础模型（2）

1. 数据初始化

以上部分我们基于timetick处理包裹，只有在整数时刻包裹才能被处理。接下来，我们考虑处理包裹的站点以流水线的方式工作，**Throughput** 的意义为一个站点能够同时处理包裹的数量，**Delay** 则

代表处理一个包裹所需要的时间，流水线则代表如果包裹 P1 正在排队且位于排队第一个，当站点内有包裹处理完时，则下一个就轮到P1直接进入处理阶段。

```
1 class Location:
2     def __init__(self, ID, pos, throughput, delay, cost):
3         self.ID = ID
4         self.pos = pos
5         self.throughput = throughput
6         self.delay = delay
7         self.cost = cost
8         self.capa = [throughput, 0]
9         self.process = []
```

- 在location中，我们仅用一个process的列表记录节点当前正在处理的以及将要处理的包裹信息，以列表形式存储[pack,开始处理时间/即将被处理的时间]

对于package,route类并没有改动，在simulator类中，我们还是选择提前计算出包裹路线并使用event这个优先队列

在这个event中包裹只有三种状态 Arrived, Processing, Send

```
1 self.route_min_time = self.chemin_time_matrix() #初始化包裹路线
2 self.events = self.init_events() #初始化event队列
```

2. 模拟过程

2.1 初始化Event

所有包裹初始状态为 Arrived，并记录创建时间，包裹信息以及路径信息

```
1 events.put([pack.time_created, pack, pack.src, pack_chemin, "ARRIVED"])
```

2.2 状态为” Arrived”

```
1 if pack_stat[-1] == "ARRIVED":
2     self.tracking_info[pack.ID].append([time, loc.ID, "ARRIVED"])
3     if len(loc.process) >= loc.capa[0]: # 如果这地方在处理的和待处理的长度大于capa
        了
4         timee = loc.delay + loc.process[-loc.capa[0]][1] # 前throughput个处
        理完的时间
```

```

5         loc.process.append([pack, timee])
6         self.events.put([timee, pack, loc.ID, chemin, "PROCESSING"])
7     else: # len(loc.process) < loc.capa 如果这地方在处理的和待处理的长小于capa
8         loc.process.append([pack, time]) # 直接加进去就可以了,因为可以即刻处理
9         self.events.put([time, pack, loc.ID, chemin, "PROCESSING"])

```


当一个包裹到达一个站点的时候, 我们的目标是找出他的Processing时间, 这个时候有两种情况

第一种情况：可以即刻处理

也就是loc.process中包裹的数量比站点的容量小, 这个时候我们直接以到达时间作为处理时间, 并相应更新loc.process 和 event

第二种情况：需要排队等待

这种情况对应loc.process中包裹的数量比站点的容量大, 这个时候包裹的预计处理时间取决于它前throughput个包裹被处理完的时间, 这个时候我们获取它前throughput个包裹的处理时间然后再加上一份delay, 我们就得到了当前包裹的预计处理时间

 假设对于节点 `s1`: Throughput=2, Delay=2

在包裹p4到达时, s1.process的情况为[[p1,2],[p2,3],[p3,4]] 那么当包裹p2被处理完时,p4才能进入被处理的阶段,也就是说p4的预计开始处理时间为 3+2=5

(这个想法基于process里的内容按处理时间升序排列, 这一点在我们更新process的过程中被保证)

2.3 状态为” Processing"

```

1 elif pack_stat[-1] == "PROCESSING":
2     self.events.put([time + loc.delay, pack, loc.ID, chemin, "SEND"])

```

在处理Processing的包裹时, 我们只需要将下一个Send的信息加入event中, 一个开始处理的包裹将在Delay这段处理时间后变成Send的状态

2.4 状态为” Send"

```

1 for k in loc.process:
2     if (k[0]).ID == pack.ID:
3         (loc.process).remove(k) # 维护loc.process 发送就代表不再是processing状态了
4         break

```

在处理Send的包裹时，最重要的一步是更新loc.process, 当包裹被发出去以后就不属于处理中或者待处理的状态了。更新完loc.process后，我们还是和之前一样，按照预设的路径走下一步，并更新event里的路径信息为 `chemin[1:]`

- 在Send状态下，我们需要加入一个分支判断是否到达目的地了，如果到达了就不往event队列里加事件了

2.5 循环结束

```
1 while not self.events.empty() : # 优先队列不为空
```

我们采取的是优先队列不为空，因为在Send部分的处理中，如果到达目的地我们是不加入事件的，而当当一个包裹送达了以后，优先队列的长度会变短，直到所有包裹均送达以后，优先队列为空，循环结束。

- 这样的处理使得我们在优先队列插入的过程所需时间更短，因为在配送过程中，优先队列长度不断变小，在相同的时间复杂度的情况下，基数n变小了

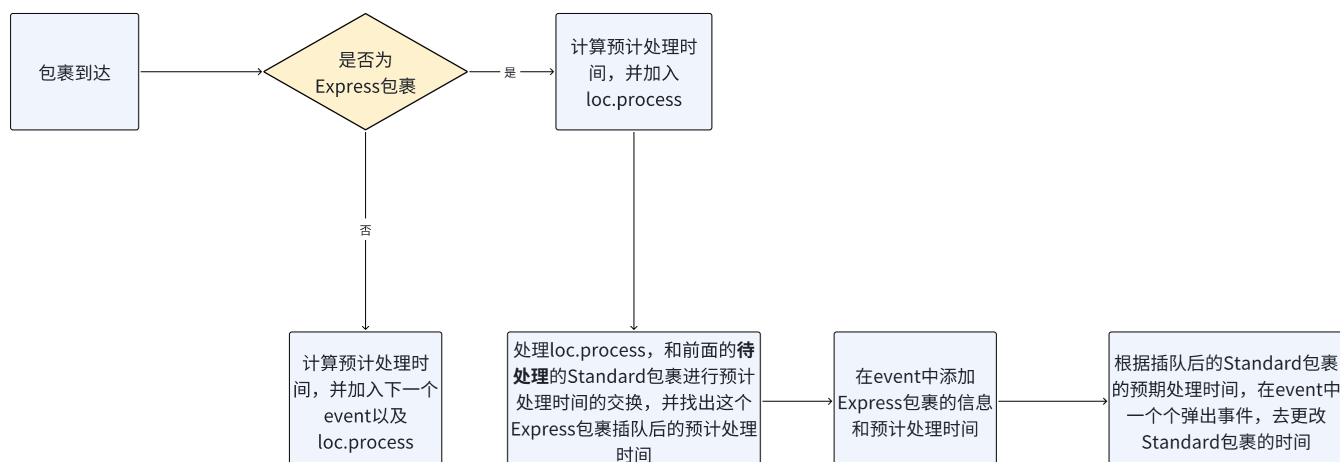
加入Queue_optimized的Event-Driven模型

1. 想法来源

由于我们的目标是实现Express包裹尽快送达，而Standard包裹只要确保他沿着我们最小代价的路径行走就可以了，因此我们考虑在Express包裹处理的过程中，加入插队的过程，也就是说当他到达一个节点时可以被优先处理

2. 模拟过程

2.1 对“Arrived” 情况的修改



1. 计算Express包裹不插队的预期处理时间，并加入loc.process中

```
1 timee = loc.delay + loc.process[-loc.capa[0]][1]
2 loc.process.append([pack, timee]) # 假设express包裹插在最后
```

2. 处理loc.process，和前面的包裹进行交换



被交换的包裹必须满足以下两条性质

- **待处理** 也就是说在loc.process中的序号必须大于Throughput
- **Standard包裹**

```
1 i = len(loc.process) - 2
2 kk = [] # 存event里需要变换的ID
3 mm = [] # 存event里需要变换的ID的新时间
4 timefinal = timee
5 rate = int((len(loc.process) - loc.capa[0])*0.5) #设定最多交换次数
6 fois = 0 #记录loc.process中的交换次数
7
8 while (loc.process[i][0]).category == 0 and i >= loc.capa[0] and fois <= rate:
9     # 逆向遍历，遇到处理中或者express的包裹则停止
10    timeexchange = loc.process[i + 1][1]
11    loc.process[i + 1][1] = loc.process[i][1]
12    mm.append(timeexchange) # mm记录standard包裹的新时间
13    timefinal = loc.process[i][1] # 最后这个express包裹落到的位置
14    loc.process[i][1] = timeexchange # 对于这个standard和express交换时间
15
16    temp = loc.process[i + 1]
17    loc.process[i + 1] = loc.process[i]
18    kk.append((loc.process[i][0]).ID) # mm记录standard包裹的编号
19    loc.process[i] = temp # 交换pack和timee整体，冒泡过程
20    i = i - 1
21    fois = fois + 1
```

变量作用：

kk：存储和Express中交换的包裹ID

mm：存储和Express中交换的包裹的新时间

rate：设定最多交换次数（防止出现Express包裹一直插队，导致Standard包裹无法处理的情况）

fois：记录Express包裹的交换次数

判断条件设定：

`(loc.process[i][0]).category == 0`：Standard包裹才能被交换

`i >= loc.capa[0]`：必须是待处理包裹

`fois <= rate`：不能超过最大交换次数

交换过程：

从后向前遍历，实现包裹之间的交换，这步的意义是找到Express包裹最终的预计处理时间，并同时维护loc.process中的事件升序排列



假设ps是Standard包裹，pe是Express包裹，pe加入后loc.process=[..., [ps,13], [pe,15]]

交换后，loc.process=[..., [pe,13], [ps,15]]

3. 在优先队列中加入Express包裹的Processing事件

```
1 self.events.put([timefinal, pack, loc.ID, chemin, "PROCESSING"])
```

4. 对我们在loc.process中更改的那些standard包裹，在event队列中做出相应更改

```
1 if i != len(loc.process) - 2: #当loc.process出现改动了我们才改动event这个队列
2     t = 0
3     pool = [] # 用来暂时存储弹出来的元素
4     while t <= timee and len(pool) < delivering-1:
5         # 只改动小于等于loc.process中最大时间的，且保证如果event全部弹出来了循环可以结
        束
6         temp = self.events.get() #不能即时放回去，否则会有死循环
7         t = temp[0] # 时间
8         packk = temp[1] # 包裹
9         cheminnew = temp[3]
10        if (temp[1].ID) in kk: # 判断是否需要改动
11            index = kk.index(temp[1].ID)
12            pool.append([mm[index], packk, loc.ID, cheminnew, "PROCESSING"])
13            pool.append(temp)
14        for ele in pool:
15            self.events.put(ele) # 放回event队列
```

变量作用：

`pool`：临时存储从event中弹出的元素

`delivering`：还需要运送个数，也就是目前event这个队列的长度

判断条件设定：

```
if i!= len(loc.process)-2:
```

- 如果在loc.process中没有发生交换，那么就不用改event（减少计算量）

```
while t <= timee and len(pool) < delivering-1:
```

- 只会弹出比loc.process中最大预计处理时间更早的event，因为只有这些里的包裹时间才可能被修改了
- 为防止所有的t均小于timee，从而全部弹出后进入死循环，我们加入第二个条件使得event为空时循环也结束

操作过程：

弹出一个事件，检查这个包裹在不在kk（需要改动的包裹列表）里，如果在，则找到它在kk中对应的序号，并用这个序号在mm（改动后的时间）中检索到对应的预计处理时间，修改事件，并将其存入pool中，最后将pool中的元素重新再加回event队列中，完成更新



Remark:

1. 我们在添加kk和mm中元素的过程中，做到了一一对应，才可以用index去调用改动后的时间
2. 用pool进行储存，而不是直接改一个就放回event的原因是优先队列每次都会弹出时间最小的一个，如果改一个放一个会导致一直弹出同一个，从而无法达到我们想要的循环效果

2.2 结果对比

我们用一次5000个包裹做试验，并于Event-Driven基础模型（2）进行对比，结果如下：

- 1 The number of faster express packages: 1319
- 2 The number of slower express packages: 5

这是我们将插队最大次数设置为 `rate = int((len(loc.process) - loc.capa[0])*0.5)` 的结果。我们可以看到，插队这个方式的效果非常理想，只有个别Express包裹会变慢，可能的原因是原本他们就可以顺利不排队经过所有节点，但因为现在有些节点Standard包裹滞后了，到达下个节点的时间也晚了，导致这些不用排队的Express包裹出现了排队的情况



变慢情况分析

原情况：

包裹p1 t=34到s1，包裹p2 t=35到s1，包裹p3 t=34.5到s1（均为Standard包裹）

包裹u1 t=36到s1（为Express包裹且u1原来在他的路径中均不用排队等待）

现情况（经过插队后）：


包裹p1 t=36到s1，包裹p2 t=36到s1，包裹p3 t=36到s1

包裹u1 t=36到s1（假设s1处容量为2，这个情况下u1到s1位置时就需要排队了，从而会导致时间变长）

Pricing初步想法

1. 场景构建

- 为每个节点设定一个向周围节点的发送间隔（interval）以及一次能发送的个数（numpack），包裹在完成处理后优先选择最近一次发送时间进行发送，如果这个发送时间的车辆已达到最大的发送数，则发送将被顺延

 比如现在s1的邻居有s2和c0

Interval = 5 : 当t为5的倍数时，才有车辆向这两个邻居发送包裹


Numpack = 2 : 一辆车最多装2个包裹

假设 t0 = 11.1 时：p1完成处理且下个节点为s2； t1 = 13.1 时：p2完成处理且下个节点为s2；

t3 = 14.1 时：p3完成处理且下个节点为s2

那么p1和p2将均在 t=15 时被发送至s2，而p3将等到 t=20 时才能被发送

- 对每个包裹计算cost，取决于这个包裹在运送路途中经过的路径以及运送过程中车辆的利用率
以下为一个具体的计算例子：

 包裹p1的路径为 [s1, s4, s7] , Routecost(s1,s4) = 3 , Routecost(s4,s7) = 4

s1->s4 : p1和三个包裹在同一辆车上从s1运往s4, 这段路径p1的代价为 $3/3=1$

s4->s7 : p1独自一个从s4运往s7, 这段路径p1的代价为 $4/1=4$

所以, p1在这种代价计算方法下,总代价为5

2. 模拟过程

2.1 对Location的初始化

- 增加了Interval,Numberpack,bustime 三个属性


```

1 self.interval = interval #发车间隔
2 self.numberpack = numberpack # 车辆容量
3 self.bustime = bustime #记录能发往哪些站点以及每个发车时刻的剩余容量

```

- 初始化的过程

```

1 def setlocation2(self, data):
2     num1 = 5 #设定station的numberpack
3     num2 = 10 #设定center的numberpack
4     inter1 = 5 #设定station的interval
5     inter2 = 10 #设定center的interval
6     nei = {}
7     .....
8
9     for i in self.neighbor(op):# 找到节点的neighbor
10         nei[i] = [num1 for k in range(int(1500 / inter1))]
11         #创建一个足够长的字典记录对于每个邻居在每个节点的车辆容量信息
12     loc = Location(op, pos, prop[0], prop[1], prop[2], num1, inter1, nei)
13     .....

```



假设s1的邻居是s2和c0, 分别对应的车辆容量为 5, 10

Loc.bustime的形式如下：

```
{ 's2': [5,5,5,5,...], 'c0': [10,10,10,10,...] }
```

2.2 对"Processing"情况的修改

在这个设定下, "Arrived"状态和"Send"状态均不需要修改, 只有由"Processing"时间加入下一个"Send"时间的这一步会受到我们规定的发车时间影响, 因此我们只需要在Event-Driven模型 (2) 的基础上修改"Processing"状态

```

1 elif pack_stat[-1] == "PROCESSING":
2     self.tracking_info[pack.ID].append([time, loc.ID, "PROCESSING"])
3     if chemin[0] == pack.dst:
4         #如果已经到目的地了, 那么这个包裹的下一个send不受车辆信息影响
5         self.events.put([time+loc.delay, pack, loc.ID, chemin, "SEND"])
6     else:
7         ui = int((time + loc.delay) / loc.interval) # ui决定包裹在什么时候被发送
8         if loc.bustime[chemin[1]][ui] > 0: #如果ui对应时间点的车辆有空位
9             self.events.put([(ui+1)*loc.interval, pack, loc.ID, chemin,
10 "SEND"])

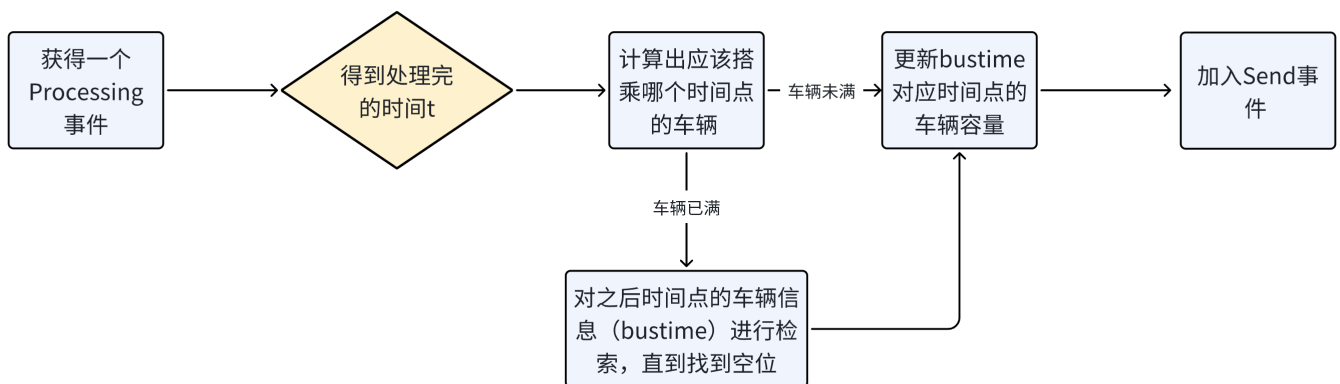
```

```

10         loc.bustime[chemin[1]][ui]-=1#容量减一
11     else:
12         #如果ui对应时间点的车辆没有空位，我们就对ui后面时间点的车辆信息进行遍历，直到找到空位
13         i = 0
14         while True:
15             i += 1
16             if loc.bustime[chemin[1]][ui+i] > 0:
17                 self.events.put([(ui + i + 1) * loc.interval, pack, loc.ID,
18 chemin, "SEND"])
19                 loc.bustime[chemin[1]][ui+i] -= 1
20                 break

```

- 处理流程



2.3 对每个包裹计算Cost

这一步的计算，我们选择在处理"Send"事件时，用同时被Send的个数以及Route cost进行计算

```

1 route_cost = self.route_cost_matrix[self.id_index(chemin[0])]
  [self.id_index(chemin[1])]
2 self.packcost[pack.ID] +=
3     route_cost / (loc.numberpack - loc.bustime[chemin[1]]
  [int(time/loc.interval+0.1)-1])

```

- `loc.bustime[chemin[1]][int(time/loc.interval+0.1)-1]`

获取bustime中这个时间点的发送信息，这个数值是本次发送的车辆剩余容量

(time/loc.interval应该为一个整数，为防止浮点数的运算误差，我们选择+0.1再取整，保证取整无误)

- `route_cost / (loc.numberpack - loc.bustime[chemin[1]] [int(time/loc.interval+0.1)-1])`

计算出本次发送路线上，包裹的Cost大小

3. 结果分析

3.1 资源浪费率 (ResourceWaste)

我们在 `loc.bustime` 中存储了每个发送时间剩余的容量，如果剩余容量小于 `Numberpack`，就代表这个时间点的车辆被发送了，我们对已发送车辆在发送时的剩余容量进行统计

```
1 base = 0
2 resourcewaste = 0
3 for loc in self.locations:
4     for oo in loc.bustime.values():
5         for ll in oo:
6             if ll < loc.numberpack: #仅对发送车辆进行统计
7                 base = base + loc.numberpack #资源总量
8                 resourcewaste = resourcewaste + ll #被浪费的资源
9 self.rateressource = resourcewaste / base
```

`base`：已发送车辆的资源量总和

`resourcewaste`：已发送车辆造成的资源浪费量的总和

`rateressource`：已发送车辆的资源浪费量与总资源量的比值

- 接下来我们将进行5次对2000个包裹的模拟,选择不同的Numberpack以及Interval,分别计算资源浪费率

 为了减少讨论变量，我们做出如下假设：

Hypothesis:

1. Numberpack: $N(\text{Center}) = 2 * N(\text{Station})$
2. Interval: $I(\text{Center}) = 2 * I(\text{Station})$

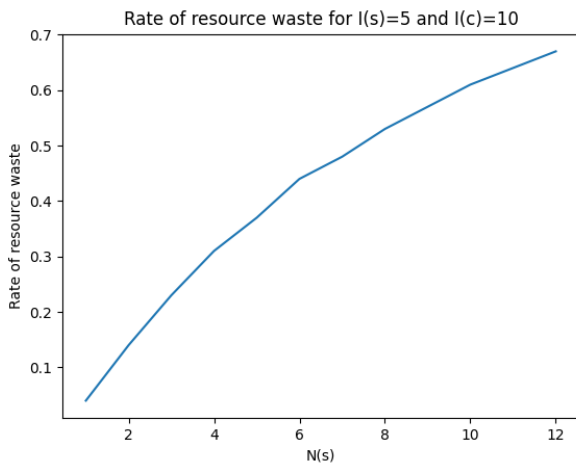


Figure1

Numberpack和资源浪费率的关系

(固定Interval分别为5和10)

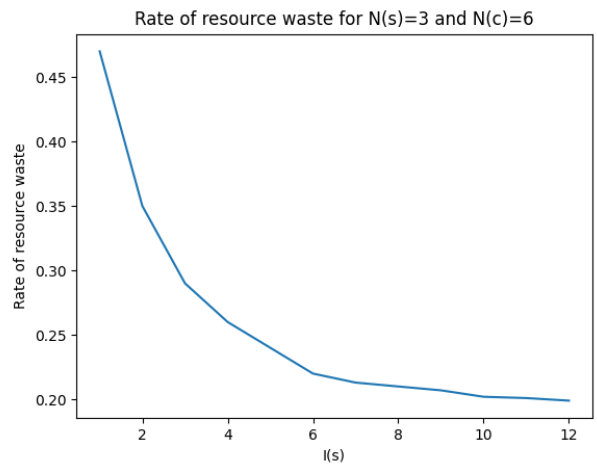


Figure2

Interval和资源浪费率的关系

(固定Numberpack分别为3和6)

● 资源浪费率的意义分析

1. 在分析Interval和资源浪费率的关系时，通过尝试更大的Interval，我们会发现最终资源浪费率不会一直减小，而是会慢慢趋近于一个定值，也就是 Figure2 图中斜率变化会更平缓，通过这个信息，我们可以知道一味的提高发车间隔不能帮助我们持续有效降低资源浪费率，但会显著影响运送时间，这点在之后与时间成本可以作一个综合考量
2. 之前的分析中，我们默认所有站点的Numberpack都被分配了相同的资源量，当我们拿到一个实际数据做出分析时，我们可以相应计算每个节点的资源浪费率，从而通过资源浪费率的信息对所有站点的资源进行再分配

```
1 d = sim.listressource
2 d_order = sorted(d.items(), key=lambda x: x[1], reverse=True)
3 print(d_order)
```



设定 $N(s)=3, N(c)=6, l(s)=5, l(c)=10$, 并对一组数据进行分析，我们得到的资源浪费率数据如下

Rate of resource waste: 0.23788350377945755

[('c3', 0.5656565656565656), ('c4', 0.4444444444444444), ('c1', 0.44339622641509435), ('c0', 0.44144144144144143), ('s1', 0.375), ('c2', 0.365296803652968), ...]

这就说明我们给Center设定的Numberpack过高，导致了Center节点的资源浪费率较高，因此为了减少总体资源浪费率，我们可以选择减少 $N(c)$

我们选择减小 $N(c)$ 为4，新数据如下：

Rate of resource waste: 0.2019245693000155

```
[('s1', 0.375), ('c3', 0.36764705882352944), ('s19', 0.36363636363636365), ('c0', 0.3541666666666667), ('s14', 0.3282051282051282), ('c4', 0.3125), ('c1', 0.2976190476190476), ('s24', 0.2777777777777778), ('s6', 0.26944444444444443), ('s16', 0.22916666666666666), ('c2', 0.2277777777777778), ...]
```

经过调整，Center处的资源浪费率显著降低

3.2 时间浪费 (TimeWaste)

`Interval` 和 `Numberpack` 除了会对资源浪费率产生影响以外，也会产生不同的时间浪费，在分析时间浪费时，我们选择和Event-Driven的基础模型进行比较，计算送达时间的偏差

```
1 x1 = sim.tracking_info      #含运送时间的模型
2 x2 = sim2.tracking_info     #基础模型
3 base2 = 0
4 timewaste = 0
5 for gg in sim.packages:
6     if x1[gg.ID][-1][0] > x2[gg.ID][-1][0]:
7         timewaste = timewaste + x1[gg.ID][-1][0] - x2[gg.ID][-1][0] #计算时间差
8         base2 = base2 + x2[gg.ID][-1][0] - x2[gg.ID][2] #基础模型的运送时间
9 print("Rate of Time waste:", timewaste/base2)
```

`timewaste` : 相比基础模型运送时间差

`base2` : 基础模型的总运送时间

`timewaste/base2` : 我们定义的时间浪费

- 接下来我们将进行5次对2000个包裹的模拟,选择不同的Numberpack以及Interval,分别计算时间浪费

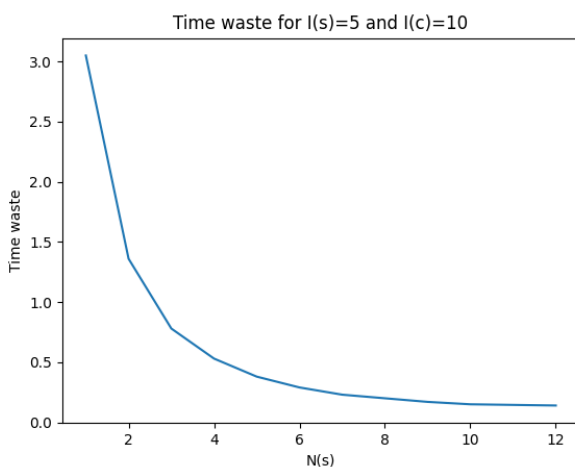


Figure1

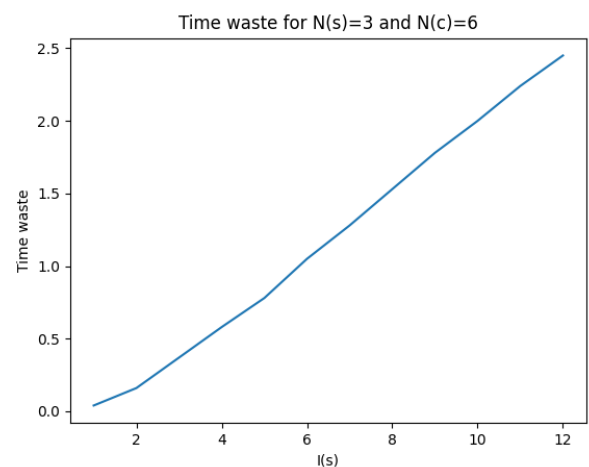


Figure2

Numberpack和时间浪费的关系

(固定Interval分别为5和10)

Interval和时间浪费的关系

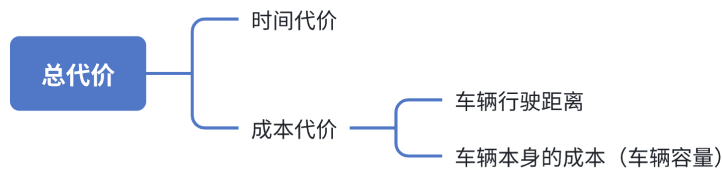
(固定Numberpack分别为3和6)

• 时间浪费的意义分析

1. 时间浪费的指标是用来衡量，在一定的 Numberpack 和 Interval 下，包裹的运送时间受到了多少的推迟
2. 在固定了 Numberpack 的情况下，我们可以看到时间浪费和 Interval 之间基本呈线性关系

3.3 总代价的计算

当我们评估一次运送过程中的代价时，分为时间代价以及成本代价，对于时间代价我们可以用定义的时间浪费来表征，而对于成本代价又分为车辆运送的距离以及车辆容量



• 成本代价的计算

在2.3中我们对每个包裹计算Cost，当我们对其求和，所得到的即是车辆行驶距离的总成本而現在，我们不仅要考虑这段车辆行驶距离所带来的成本，还要考虑车辆容量所带来的成本

```
1 self.packcost[pack.ID] +=  
2     (1+0.1*loc.numberpack)* route_cost/(loc.numberpack-loc.bustime[chemin[1]]  
    [int(time/loc.interval+0.1)-1])
```

$(1+0.1*loc.numberpack)$ ：取决于车辆容量成本到底对代价有多大的影响（在这里我们以这样的等式模拟）

例子：

从s1到s7路径成本为2

情况一：包裹1搭乘容量为3的车辆被运输到s7，车辆上共载有2个包裹

$$\text{Cost} = (1+0.1*3)*2/2$$

情况二：包裹1搭乘容量为4的车辆被运输到s7，车辆上共载有1个包裹

$$\text{Cost} = (1+0.1*4)*2/1$$

意义：

有了时间牺牲和资源牺牲的数值，我们就可以通过所需要的时间与代价之间的对应关系（需按照实际需要），改变Interval和Numpack去最小化整体代价。

使用指引

共有五个文件夹，其中四个各对应一个模型，其中：Basic文件夹下是Time-Driven模型，Event-Driven-1文件夹下是Event-Driven-1的基础模型，Route_optimize文件夹下是加入Route_optimized的Event-Driven模型，Queue文件夹下是加入Queue_optimized的Event-Driven模型，cost文件夹下是优化了pricing的Event-Driven模型。

1. 在前四个文件夹中都有一个data_gen.py和simulator.py，请先打开一个文件夹，然后打开这两个代码文件，在确认两者的"packet_num"一致后，请运行simulator.py文件。
2. 你会看到依次看到以下输出：

```
1 Data import —— Succeeded
2 Packages, locations and routes information is saved in csv files.
```

该输出说明数据已经成功导入进入模拟流程，站点、路径和包裹信息已被保存在同路径下的csv文件中。

```
1 Package dealing —— Succeeded
2 The tracking information is saved in tracking_info.csv.
```

该输出说明模拟流程已经顺利结束，并且包裹的tracking_info已经被保存在同路径下的tracking_info.csv中。

tracking_info中存储的数据格式如下：（包裹ID，始发站，终点站，创建时间，抵达时间，Event_Log）

ID	Src	Dst	TimeCreat	TimeDeliv	Log
309dcde3	s4	s21	0.000285	607.1721	[[0.000284
ee61324a	s18	s13	0.000571	224.8792	[[0.000571
7d4ef53c	s21	s23	0.000626	645.9928	[[0.000626
d4a58424	s5	s15	0.000751	948.041	[[0.000751
1f4c82ee	s10	s24	0.000965	1503.743	[[0.000965
fc95ddae	s15	s21	0.001078	1157.897	[[0.001078
e723ddd8	s23	s12	0.00125	933.731	[[0.001250
a524eddc	s2	s24	0.001316	129.6248	[[0.001316
5644b9cb	s23	s19	0.001322	873.1211	[[0.001321
45be80cc	s15	s5	0.001768	1127.042	[[0.001767

3. 当你看到以下输出时，如果你希望查询包裹的相关信息，请键入Y，否则N。如果你输入Y，请继续第4步；否则，请跳转第10步。

Do you want to check package's info? (Y/N)

4. 当你看到以下输出时，请输入你想查询的包裹的UUID（可以在packets.csv里查询UUID），如果包裹未找到，则会弹出"Package not found, please enter another UUID: "，这时请你重新输入一个UUID；如果包裹可以查询到，可以继续第5步。

Which package do you want to check? (Enter a UUID)

5. 当你看到以下输出时，请输入你想查询的时间，程序会输出该时间下，该包裹的状态，包括当前路径、坐标以及状态。当完成这步后，请继续第6步。

At what time? (Enter a number)

6. 当你看到以下输出时，如果你希望继续查询不同时间，请键入Y；否则N。如果你键入Y，请继续第7步；否则，跳转第10步。

Do you want to check another time? (Y/N)

7. 请再次输入一个时间，程序会输出该时间下，该包裹的状态，包括当前路径、坐标以及状态。当完成这步后，请跳转第8步。

Enter a time:

8. 当你看到以下输出时，如果你希望看到该包裹的动态路径，请键入Y；否则N。如果你键入Y，则会弹出一张动图，展示包裹的路径，**当你把图像关闭后，请继续第9步**；否则，请直接跳转第9步。

Do you want to draw its route? (Y/N)

9. 当你看到以下输出时，如果你希望继续查询不同包裹，请键入Y；否则N。如果你键入Y，请跳转第4步；否则，请继续第10步。

Do you want to check another package? (Y/N)

10. 当你看到以下输出时，如果你希望看到整个地图以及Time和Cost的加权图，请键入Y，否则N。如果你输入Y，则会看到两张图像，**当你把他们关闭后**，可以继续第11步；如果输入N，请跳转第11步。

Do you want to draw Cost and Time Graph of Routes? (Y/N)

11. 当你看到以下输出时，如果你希望查询经过某站点的包裹ID，请键入Y，否则N。如果你键入Y，则继续第12步；否则，程序将会结束，并返回"Simulation terminated."。

Do you what to check location's pass-by packages? (Y/N)

12. 当你看到以下输出时，请键入一个站点的ID（例如s2, c3），程序会将该站点经过包裹的信息存储在同路径下的一个"loc_+ID"的csv文件中。请继续第13步。

Which location do you want to check? (Enter an ID of a station)

13. 当你看到以下输出时，如果你希望查询其他站点，请键入Y，否则N。如果你键入Y，请跳转第12步；否则，程序将会结束，并返回"Simulation terminated."。

Do you want to check another location? (Y/N)

- 在最后一个文件夹(Cost)里，是关于代价评估以及对应情景模拟的代码，`simulator.py` 中的 `simulator` 是加入发车间隔和车辆容量的实现，`data.json` 是我们处理时所用的数据，输出是路径总代价，资源浪费率，时间浪费以及各节点资源浪费率，如果想具体看 `trackinginfo` 可以用最后一行的 `print(sim.trackinginfo)`